Y Combinator,
CC/CK machine,
and continuations

CS245 — Spring 2019

Authors: Kris Micinski + Thomas Gilray

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
x ::= <vars>
```

Infinity

Ω

((λ (x) (x x)) (λ (x) (x x)))

Key: U takes a function and calls it on itself

Ω

```
((λ (x) (x x)) (λ (x) (x x)))
```

U

```
(define U (λ (f) (f f)))
```

```
(letrec ([fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1)))))])
  (fib 3))
```

```
(let ([fib (U (lambda (f)
               (lambda (x) (if (= x 0) 1 (* x (... (- x 1)))))))])
  (fib 3))
```

What can I type right here to make fib work?

(Hint: the answer can be written in 5 characters)

```
(define U (λ (f) (f f)))
```

```
(letrec ([fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1)))))])
  (fib 3))
```

```
(let ([fib (U (lambda (f)
               (lambda (x) (if (= x 0) 1 (* x (... (- x 1))))))))])
  (fib 3))
```

What can I type right here to make fib work?

(f f)

Y combinator

```
(letrec ([fact (λ (n)
                 (if (= n 0)
                     1
                     (* n (fact (- n 1)))))])
  (fact 5))
```

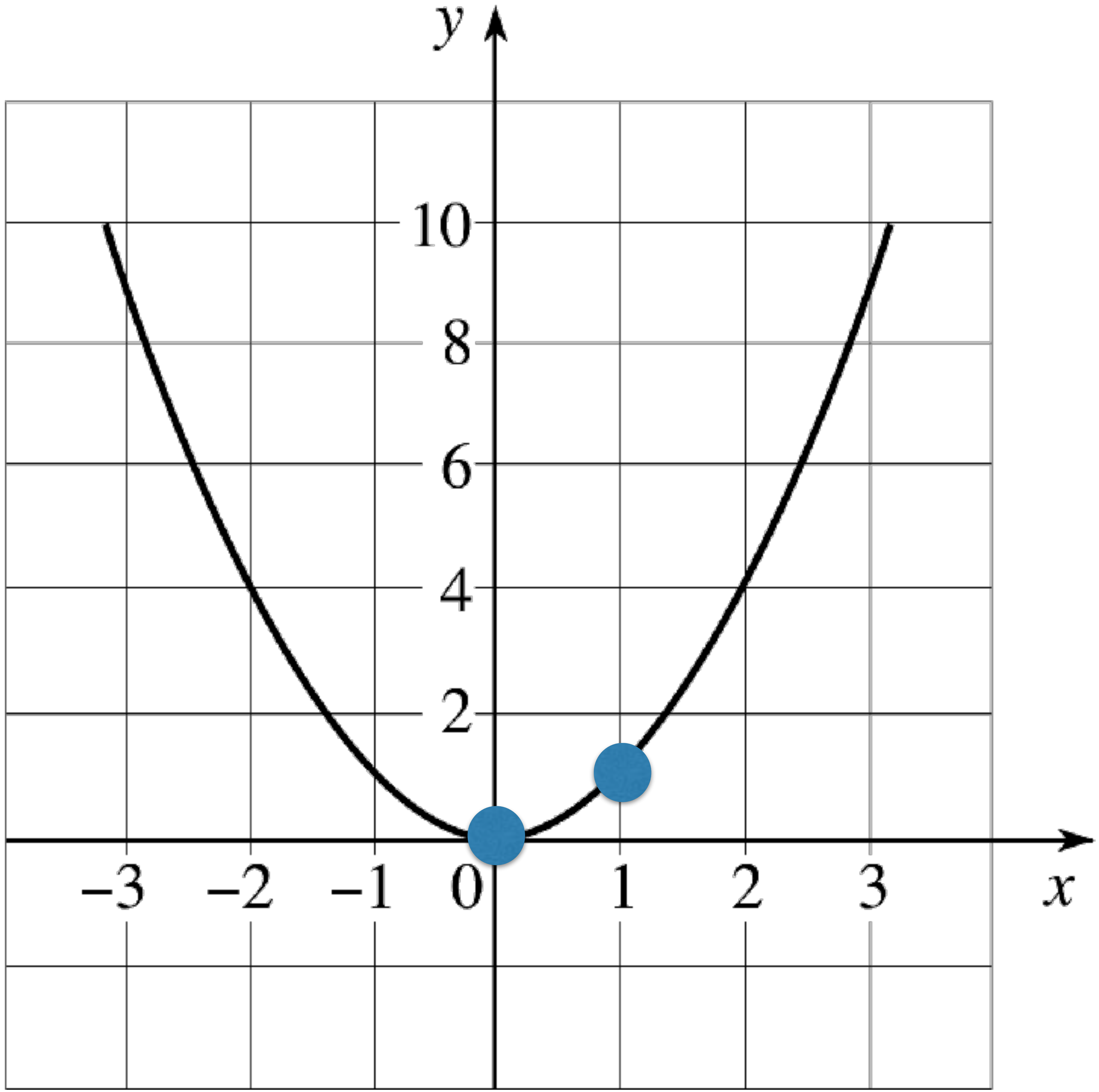
Key idea: instead of

```
(let ([mk (λ (mk) (λ (n)
                    (if (= n 0)
                        1
                        (* n ((mk mk) (- n 1))))))])
  ((mk mk) 5))
```

Y

$$(Y\ f) = f\ (Y\ f)$$

(It's a fixed-point combinator!)


y
10
8
6
4
2
–3
–2
–1
0
1
2
3
x

Three step process for deriving Y

```
(Y f) = f (Y f)
```

```
Y = (λ (f) (f (Y f)))
```

1. Treat as definition

```
mY = (λ (mY)
       (λ (f)
         (f ((mY mY) f))))
```

2. Lift to mk-Y,
use self-application

```
mY = (λ (mY)
       (λ (f)
         (f (λ (x) (((mY mY) f) x)))))
```

3. Eta-expand

U-combinator: `(U U)` **is Omega**

```
Y  =  (U (λ (y) (λ (f)
           (f (λ (x) (((y y) f) x))))))
```

```
(let ([fact (Y (λ (fact) (λ (n)
                  (if (= n 0)
                      1
                      (* n (fact (- n 1))))))])
  (fact 5))
```

Try an example!!!

```
(define Y ((λ (x) (x x)) (λ (y) (λ (f)
                                    (f (λ (x) (((y y) f) x)))))))
```

```
(define (fib x)
  (if (or (= x 0) (= x 1))
      1
      (+ (fib (- x 1)) (fib (- x 2)))))
```

Rewrite this to use the Y combinator instead

```
e ::= (lambda (x) e)
    | (e e)
    | x
```

De-churching

```
(define (church->nat cv)
                      )


(define (church->list cv)






                            )



(define (church->bool cv)

                         )
```

De-churching

```
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)





                         )



(define (church->bool cv)

                      )
```

De-churching

```
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)
        ((cv (λ (car)
               (λ (cdr)
                 (cons car
                   (church->list cdr)))))
         (λ (na) ‘())))


(define (church->bool cv)

                          )
```

De-churching

```
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)
        ((cv (λ (car)
               (λ (cdr)
                 (cons car
                   (church->list cdr)))))
         (λ (na) ‘())))


(define (church->bool cv)
        ((cv (λ () #t))
         (λ () #f)))
```

```
(letrec ([map (λ (f lst)
                (if (null? lst)
                    '()
                    (cons (f (car lst))
                          (map f (cdr lst)))))])

  (map (λ (x) (+ 1 x))
       '(0 5 3)))
```

```
(define lst
  ((((((((((λ (Y-comb)
            (λ (church:null?)
              (λ (church:cons)
                (λ (church:car)
                  (λ (church:cdr)
                    (λ (church:+)
                      (λ (church:*)
                        (λ (church:not)
                          ((λ (map)
                             ((map
                               (λ (x)
                                 ((church:+ (λ (f) (λ (x) (f x))))
                                  x)))
                              ((church:cons (λ (f) (λ (x) x)))
                               ((church:cons
                                 (λ (f)
                                   (λ (x) (f (f (f (f (f x))))))))
                                ((church:cons
                                  (λ (f) (λ (x) (f (f (f x))))))
                                 (λ (when-cons)
                                   (λ (when-null)
                                     (when-null (λ (x) x)))))))))
                           (Y-comb
                            (λ (map)
```

```
> (map church->nat (church->list lst))
'(1 6 4)
```

Abstract Machine Zoo

C Term-rewriting Machine

Evaluation contexts

Restrict the order in which we may simplify a program's redexes

$$
\begin{aligned}
\mathcal{E} \; ::= \; & (\mathcal{E}\ \ e) \\
| \; & (v\ \ \mathcal{E}) \\
| \; & \square
\end{aligned}
$$

(left-to-right) CBV evaluation

$$
\begin{aligned}
\mathcal{E} \; ::= \; & (\mathcal{E}\ \ e) \\
| \; & \square
\end{aligned}
$$

(left-to-right) CBN evaluation

```
v  ::=  (λ (x) e)

e  ::=  (λ (x) e)
     |  (e e)
     |  x
```

Context and redex

For CBV a redex must be `(v v)`
For CVN, a redex must be `(v e)`

$$\mathcal{E}[\overbrace{\texttt{(v v)}}^{\texttt{r}}] =$$

```
(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (w) w))
```

$\mathcal{E}$ `= (□ (λ (w) w))`

`r = ((λ (x) ((λ (y) y) x)) (λ (z) z))`

Context and redex

$\mathscr{E}$[r] =

(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (w) w))

$\mathscr{E}$ = (□ (λ (w) w))

r = ((λ (x) ((λ (y) y) x)) (λ (z) z))

$\rightarrow_\beta$ ((λ (y) y) (λ (z) z))

Put the reduced redex back in its evaluation context…

$\mathscr{E}$ = `(□ (λ (w) w))`

`r = ((λ (x) ((λ (y) y) x)) (λ (z) z))`

$\to_\beta$ `((λ (y) y) (λ (z) z))`

$\downarrow$ $\mathscr{E}$`[r]`

`(((λ (y) y) (λ (z) z)) (λ (w) w))`

Exercises—can you evaluate…

```
1)   (((λ (y) y) (λ (z) z)) (λ (w) w))


2)   ((λ (u) (u u)) (λ (x) (λ (x) x)))


3)   (((λ (x) x) (λ (y) y))
      ((λ (u) (u u)) (λ (z) (z z))))
```

Abstract Machine Zoo

C	Term-rewriting Machine
CC	Context and Redex Machine
CK	Control / Continuation Machine

Next time…

Continuations: first-class control

Continuations

A ***continuation*** is a return point, a call stack, or the remainder of the program, viewed as a function.

In Scheme, continuations are first-class values that can be captured using the language form `call/cc` and passed around to be invoked later.

First-class continuations

We may consider several alternative viewpoints on first-class continuations:

A ***continuation*** is a value encoding a *saved return point* to resume.

A ***continuation*** is a function encoding the *remainder of the program*.

A ***continuation*** is a function that never returns. When invoked on an input value, it resumes a previous return point with that value, and finishes the program from that return point until it exits.

Continuations generalize all known control constructs: gotos, loops, return statements, exceptions, C's `longjmp`, threads/coroutines, etc

Continuations

Continuations are said to permit ***time travel***, in the sense that they permit jumping back to a saved dynamic *evaluation context*: a *previous* call stack, or point in time, or a *future* one!

This may shorten the stack

…grow the stack

…or replace it entirely.

`(call/cc e0)`

call with current continuation

`call/cc` takes a single argument, a callback, which it applies on the ***current continuation***—that is, the return from call/cc as a first-class function that saves the full call stack under call/cc.

```
(+ 1 (call/cc (lambda (k) (k 2))))
;; =>  3
```

Takes the call stack at the second argument expression of `(+ ...)` and saves it, essentially as a function, bound to `k`, that can be invoked on a value for that expression at a later point in time.

When k is invoked on the number `2`, execution jumps back to the saved return point for `call/cc` and returns `2`, returning `3` from the program as a whole.

```
(+ 1 (call/cc (lambda (k) (k 2))))
;; => 3
```

The program never returns from call `(k 2)` because ***undelimited continuations*** run until the program exits.

`call/cc` gives us undelimited (a.k.a. full) continuations.

```
(+ 1 (call/cc (lambda (k) (k 2) (print 0))))
;; => 3        (print 0) is never reached
```

```
(+ 1 (call/cc (lambda (k) (k 2))))
;; => 3
```

This `call/cc`'s behavior is *roughly* the same as the application:

```
((lambda (k) (k 2))
 (lambda (n) (exit (print (+ 1 n)))))
;; => 3
```

Where the high-lit continuation `(lambda (n) …)` takes a return value for the `(call/cc …)` expression and finishes the program.

```
(let ([cc (call/cc (lambda (k) k))])
  ...)
```

A common idiom for `call/cc` is to let-bind the current continuation.

```
(let ([cc (call/cc (lambda (k) k))])
  ...)
```

Note that applying call/cc on the identity function is exactly the same as applying it on the u-combinator!

```
(let ([cc (call/cc (lambda (k) (k k)))])
  ...)
```

Why is this the case?

`call/cc` makes a tail call to `(lambda (k) …)`, so the body of the function is the same return point as the captured continuation k!

```
(let ([cc (call/cc (lambda (k) k))])
   ...)
```

This return point …is the same as this one…

```
(let ([cc (call/cc (lambda (k) (k k)))])
   ...)
```

…and calling k on itself, returns k to itself!

Returning value v is the same as *calling* that saved return point *on* v.

```
(let ([cc (call/cc (lambda (k) k))])

   ;; loop body goes here

   (if (jump-to-top?)
       (cc cc)
       return-value))
```

Continuations can be used to jump back to a previous point.

Just as we could have invoked `call/cc` on the u-combinator, to jump back to the let-binding of `cc`, returning `cc`, we call `(cc cc)`.

```
(define (fun x)

  (let ([y (if (p? x)
               …
               …)])

    (g x y)))
```

A simple use of continuations is to implement a ***preemptive return***.

What if we wanted to return from `fun` within the right-hand-side of the `let` form?

Binds the return-point of the current call to `fun` to a continuation `return`.

```
(define (fun x)
  (call/cc (lambda (return)

    (let ([y (if (p? x)
                 ...
                 (return x))])

      (g x y)))))
```

Uses the continuation `return` to jump back to the return point of `fun` and yield value `x` instead of binding `y` and calling `g`.

Try an example. What do each of these 3 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 3)))))))
```

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 (k1 3))))))))
```

```
(call/cc (lambda (k0)
           (+ 1
              (call/cc (lambda (k1)
                         (+ 1 (k1 3))))
              (k0 1))))
```

Try an example. What do each of these 3 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 3)))))))
```

3

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 (k1 3))))))))
```

4

```
(call/cc (lambda (k0)
           (+ 1
              (call/cc (lambda (k1)
                         (+ 1 (k1 3))))
              (k0 1))))
```

1

Continue and break

A Python `while` loop on the left that supports `continue` and `break` can be implemented using `call/cc` as the Scheme on the right.

```
while cond:
  body
else:
  otherwise
```

```
(call/cc (λ (break)
  (letrec ([loop (λ ()
                (when cond
                   (call/cc (λ (continue)
                               body))
                   (loop)))])
   (loop)
   otherwise)))
```

Continuations and mutation

```
(let* ([n 2]
       [cc (call/cc (lambda (k) k))])
  (set! n (+ n 1))
  (if (<= n 4)
      (cc cc)
      n))
```

Does this program terminate? What does it return?

Continuations and mutation

```
(let* ([n 2]
       [cc (call/cc (lambda (k) k))])
  (set! n (+ n 1))
  (if (<= n 4)
      (cc cc)
      n))
```

This loop terminates and returns 5.

This illustrates that invoking a continuation resumes a previous call stack, but *does not* revert mutations—changes made in the heap.

Try an example. What do each of these 2 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(define n 3)
(+ n (call/cc
        (lambda (cc)
          (set! n (+ n 1))
          (cc 1))))
```

```
(define n 3)
(+ (call/cc
        (lambda (cc)
          (set! n (+ n 1))
          (cc 1)))
   n)
```

Try an example. What do each of these 2 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(define n 3)
(+ n (call/cc
       (lambda (cc)
         (set! n (+ n 1))
         (cc 1))))
```

4

```
(define n 3)
(+ (call/cc
       (lambda (cc)
         (set! n (+ n 1))
         (cc 1)))
   n)
```

5

Stack-passing (CEK) semantics

(implementing first-class continuations)

C **Control-expression**

Term-rewriting / textual reduction

Context and redex for deterministic eval

CE **Control & Env machine**

Big-step, explicit closure creation

CES **Store-passing machine**

Passes addr->value map in evaluation order

CEK **Stack-passing machine**

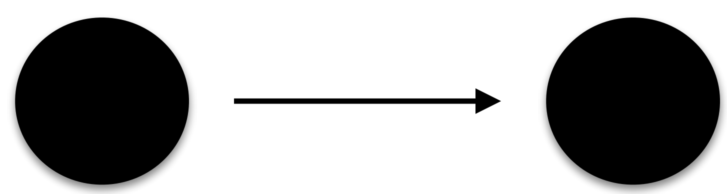
Passes a list of stack frames, small-step

$$\frac{(\texttt{e}_0, \text{env}) \Downarrow (\texttt{(λ (x) e}_2\texttt{)}, \text{env'}) \qquad (\texttt{e}_1, \text{env}) \Downarrow v_1 \qquad (\texttt{e}_2, \text{env'}[x \mapsto v_1]) \Downarrow v_2}{(\texttt{(e}_0\ \texttt{e}_1\texttt{)}, \text{env}) \Downarrow v_2}$$

$$\frac{}{(\texttt{(λ (x) e)}, \text{env}) \Downarrow (\texttt{(λ (x) e)}, \text{env})}$$

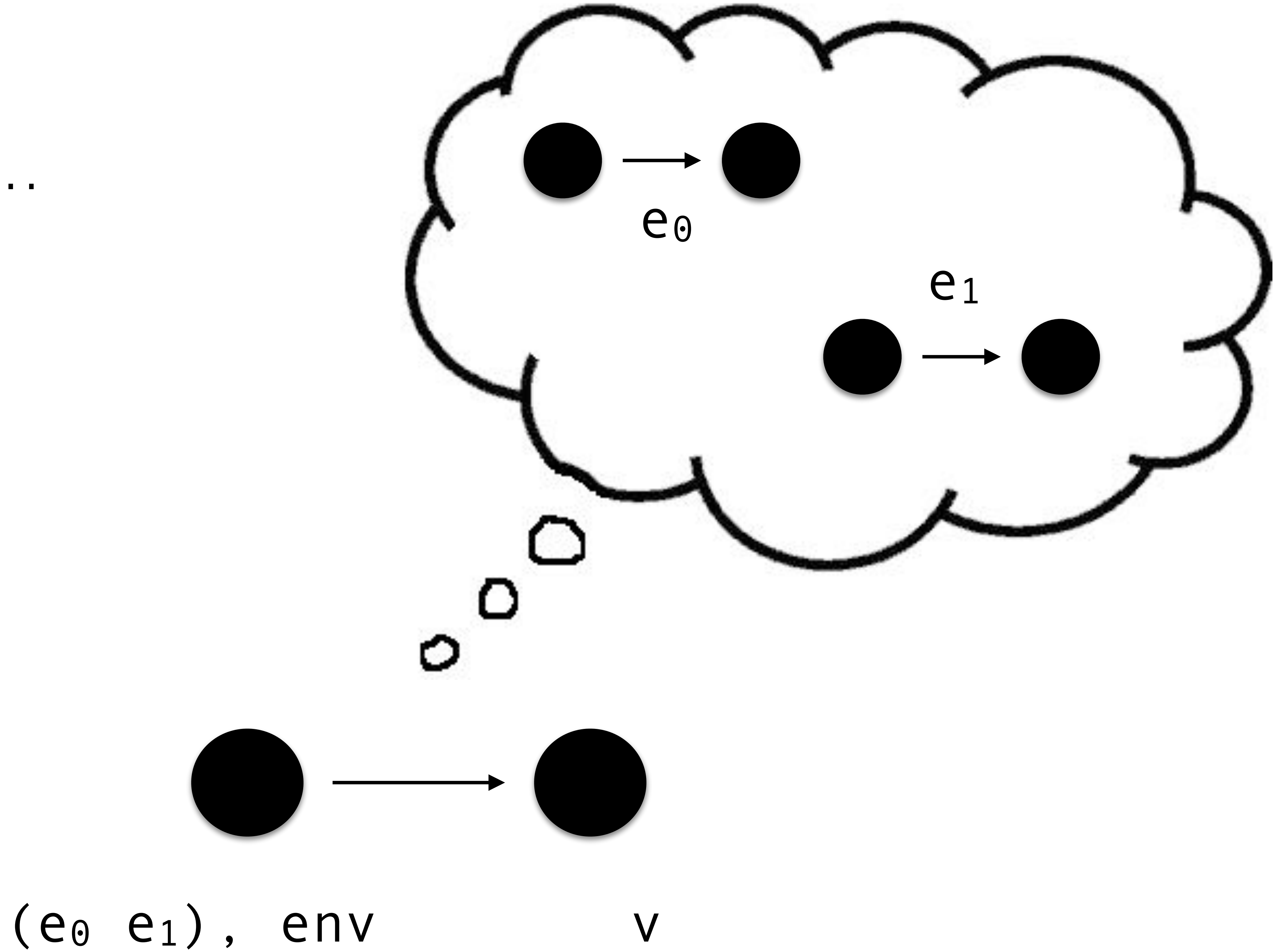
$$\frac{}{(\texttt{x}, \text{env}) \Downarrow \text{env}(\texttt{x})}$$

Previously…



`(e₀ e₁), env` → `v`

Previously…



```
(define (interp e env)
  (match e
         [(? symbol? x)
          (hash-ref env x)]

         [`(λ (,x) ,e0)
          `(clo (λ (,x) ,e0) ,env)]

         [`(,e0 ,e1)
          (define v0 (interp e0 env))
          (define v1 (interp e1 env))
          (match v0
            [`(clo (λ (,x) ,e2) ,env)
             (interp e2 (hash-set env x v1))])]))
```

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

k ::= **halt** | **ar**(e, env, k)
| **fn**(v, k)

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

k ::= **halt** | **ar**(e, env, k)

| **fn**(v, k)

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

$\mathcal{E}$ ::= ($\mathcal{E}$ e)

| (v $\mathcal{E}$)

| □

$$((\mathtt{e}_0\ \mathtt{e}_1), \mathrm{env}, k) \rightarrow (\mathtt{e}_0, \mathrm{env}, \mathbf{ar}(\mathtt{e}_1, \mathrm{env}, k))$$

$$(\mathtt{x}, \mathrm{env}, \mathbf{ar}(\mathtt{e}_1, \mathrm{env}_1, k_1)) \rightarrow (\mathtt{e}_1, \mathrm{env}_1, \mathbf{fn}(\mathrm{env}(\mathtt{x}), k_1))$$

$$((\lambda\ (\mathtt{x})\ \mathtt{e}), \mathrm{env}, \mathbf{ar}(\mathtt{e}_1, \mathrm{env}_1, k_1)) \rightarrow (\mathtt{e}_1, \mathrm{env}_1, \mathbf{fn}(((\lambda\ (\mathtt{x})\ \mathtt{e}), \mathrm{env}), k_1))$$

$$(\mathtt{x}, \mathrm{env}, \mathbf{fn}(((\lambda\ (\mathtt{x}_1)\ \mathtt{e}_1), \mathrm{env}_1), k_1)) \rightarrow (\mathtt{e}_1, \mathrm{env}_1[\mathtt{x}_1 \mapsto \mathrm{env}(\mathtt{x})], k_1)$$

$$((\lambda\ (\mathtt{x})\ \mathtt{e}), \mathrm{env}, \mathbf{fn}(((\lambda\ (\mathtt{x}_1)\ \mathtt{e}_1), \mathrm{env}_1), k_1)) \rightarrow (\mathtt{e}_1, \mathrm{env}_1[\mathtt{x}_1 \mapsto ((\lambda\ (\mathtt{x})\ \mathtt{e}), \mathrm{env})], k_1)$$

call/cc semantics

$$(\texttt{(call/cc (}\lambda\texttt{ (x) e}_0\texttt{))}, env, k) \rightarrow (\texttt{e}_0, env[x \mapsto k], k)$$

$$(\texttt{((}\lambda\texttt{ (x) e}_0\texttt{)}, env, \mathbf{fn}(k_0, k_1)) \rightarrow (\texttt{((}\lambda\texttt{ (x) e}_0\texttt{)}, env, k_0)$$

$$(\texttt{x}, env, \mathbf{fn}(k_0, k_1)) \rightarrow (\texttt{x}, env, k_0)$$

$$e ::= \ldots \mid (\texttt{let}\ ([x\ e_0])\ e_1)$$

$$k ::= \ldots \mid \textbf{let}(x, e, env, k)$$

$$(x, env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$$

$$((\lambda\ (x)\ e), env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto ((\lambda\ (x)\ e), env)], k_1)$$

$$e \ ::= \ \ldots$$

$$\mid \ \texttt{(let ([x e}_0\texttt{]) e}_1\texttt{)}$$

$$(\texttt{x}, \text{env}, \textbf{fn}((\texttt{(λ (x}_1\texttt{) e}_1\texttt{)}, \text{env}_1), k_1)) \ \rightarrow \ (\texttt{e}_1, \text{env}_1[x_1 \mapsto \text{env}(x)], k_1)$$

$$(\texttt{(λ (x) e)}, \text{env}, \textbf{fn}((\texttt{(λ (x}_1\texttt{) e}_1\texttt{)}, \text{env}_1), k_1)) \ \rightarrow \ (\texttt{e}_1, \text{env}_1[x_1 \mapsto (\texttt{(λ (x) e)}, \text{env})], k_1)$$

$$k \ ::= \ \ldots \ \mid \ \textbf{let}(x, e, \text{env}, k)$$

These are nearly identical because a let form is just an immediate application of a lambda!

$$(\texttt{x}, \text{env}, \textbf{let}(x_1, \texttt{e}_1, \text{env}_1, k_1)) \ \rightarrow \ (\texttt{e}_1, \text{env}_1[x_1 \mapsto \text{env}(x)], k_1)$$

$$(\texttt{(λ (x) e)}, \text{env}, \textbf{let}(x_1, \texttt{e}_1, \text{env}_1, k_1)) \ \rightarrow \ (\texttt{e}_1, \text{env}_1[x_1 \mapsto (\texttt{(λ (x) e)}, \text{env})], k_1)$$

CEK-machine evaluation

$$
\begin{aligned}
(\mathtt{e_0}, [\,], ()) &\to \ldots \\
&\to \ldots \\
&\to \ldots \\
&\to \ldots \\
&\to (x, env, \textbf{halt}) \to env(x)
\end{aligned}
$$

consider the following question.

Is it possible to take an arbitrary Racket/Scheme program and transform it systematically so that no function ever returns?